Using Cray Performance Analysis Tools

S–2376–50

Version 3.1 Published October 2006 First release. Supports CrayPat 3.1 and Cray Apprentice2 3.1 running on Cray XT systems.

Version 4.1 Published December 2007 Supports CrayPat 4.1 and Cray Apprentice2 4.1 running on Cray XT3, Cray XT4, and Cray XT5 systems, including Cray $XT5_h$ systems with Cray X2 compute blades.

Version 5.0 Published September 2009 Supports CrayPat and Cray Apprentice2 5.0 release running on Cray XT systems, excluding Cray $XT5_h$ (Cray X2) systems.

New Features

Using Cray Performance Analysis Tools S–2376–50

This guide is a complete rewrite of *Using Cray Performance Analysis Tools*. It supports the 5.0 release of CrayPat and Cray Apprentice2. Major changes from earlier versions include:

- Initial documentation of Automatic Program Analysis. See Using Automatic Program Analysis on page 21.
- Expanded documentation of `pat_build` environment variables and build directives. See Advanced Users: Environment Variables and Build Directives on page 22.
- Expanded documentation of the CrayPat OpenMP API. See Advanced Users: OpenMP on page 29.
- Expanded documentation of CrayPat run time environment variables. See Chapter 3, Using the CrayPat Run Time Environment on page 31.
- Expanded documentation of the `pat_help` online help system, including the new Frequently Asked Questions (FAQ) feature. See Online Help on page 13.
- Initial documentation of the new Cray Apprentice2 online help system. See Online Help on page 15.
- Initial documentation of CrayPat and Cray Apprentice2 data file compatibility issues. See Upgrading from Earlier Versions on page 14.
- The deprecated commands `pat_hwpc` and `pat_run` are removed in this release.
- Support for Cray XT systems running the Catamount compute node operating system and Cray $XT5_h$ systems equipped with Cray X2 compute nodes is removed in this release.

Contents

Page

Page

Procedures

Tables

Figures

Introduction [1]

The Cray Performance Analysis Tools are a suite of optional utilities that enable you to capture and analyze performance data generated during the execution of your program on a Cray XT system. The information collected and analysis produced by use of these tools can help you to find answers to two fundamental programming questions: *How fast is my program running?* and *How can I make it run faster?*

The Cray Performance Analysis Tools suite consists of two separately licensed components:

- **CrayPat**: the program instrumentation, data capture, and basic text reporting tool
- **Cray Apprentice2**: the graphical analysis and data visualization tool

This guide is intended for programmers and application developers who write, port, or optimize software applications for use on Cray XT systems running the Cray Linux Environment (CLE) operating system. We assume you are already familiar with the Cray XT development and execution environments and the general principles of program optimization, and that your application is already fully debugged and capable of running to planned termination. If you need more information about the Cray XT development and execution environments or about debugging applications, see the *Cray XT Programming Environment User's Guide*.

A discussion of massively parallel programming optimization techniques is beyond the scope of this guide.

> **Note:** The Cray Performance Analysis Tools 5.0 release does not support Cray XT systems running the Catamount compute node operating system, nor does it support Cray $XT5_h$ systems equipped with Cray X2 compute nodes.

Cray XT systems feature a variety of processors and support a variety of compilers. Because of this, your results may vary from the examples discussed in this guide. Most of the examples in this guide were developed using the Cray Compiling Environment (CCE) 7.1 compilers on a Cray XT4 system with quad-core processors.

1.1 Analyzing Program Performance with CrayPat

The performance analysis process consists of three basic steps.

1. **Instrument** your program, to specify what kind of data you want to collect under what conditions.
2. **Execute** your instrumented program, to generate and capture the desired data.
3. **Analyze** the resulting data.

Accordingly, CrayPat consists of the following major components:

- `pat_build`, the utility used to instrument programs
- the CrayPat run time environment, which collects the specified performance data
- `pat_report`, the first-level analysis tool used to produce text reports or export data for more sophisticated analysis
- `pat_help`, the command-line driven online help system

All CrayPat components, including the man pages and help system, are available only when the CrayPat module is loaded.

1.1.1 Loading CrayPat and Compiling

To use CrayPat, first load your programming environment of choice, and then load the CrayPat module.

```
> module load xt-craypat
```

For successful results, the CrayPat module must be loaded before you compile the program to be instrumented, instrument the program, execute the instrumented program, or generate a report. If you want to instrument a program that was compiled before the CrayPat module was loaded, you may under some circumstances find that re-linking it is sufficient, but as a rule it's best to load the CrayPat module and then recompile.

When instrumenting a program, CrayPat requires that the object (`.o`) files created during compilation be present, as well as the library (`.a`) files, if any. However, most compilers automatically delete the `.o` and `.a` files when working with single source files and compiling and linking in a single step, therefore it is good practice to compile and link in separate steps and use the compiler command line option to preserve these files. For example, if you are using the Cray Compiling Environment (CCE) Fortran compiler, compile using either of these command line options:

```
> ftn -c sourcefile.f
```

Alternatively:

```
> ftn -h keepfiles sourcefile.f
```

Then link the object files to create the executable program:

```
> ftn -o executable sourcefile.o
```

For more information about compiling and linking, see your compiler's documentation.

1.1.2 Instrumenting the Program

After the CrayPat module is loaded and the program is compiled and linked, you can instrument your program for performance analysis experiments. This is done using the `pat_build` command. In simplest form, it is used like this:

```
> pat_build executable
```

This produces a copy of your original program, which is named *executable*`+pat` (for example, `a.out+pat`) and instrumented for the default experiment. Your original executable remains untouched.

The `pat_build` command supports a large number of options and directives, including an API that enables you to instrument specified regions of your code. These options and directives are documented in the `pat_build`(1) man page and discussed in Chapter 2, Using `pat_build` on page 17.

The CrayPat API is discussed in Advanced Users: The CrayPat API on page 25.

1.1.2.1 Automatic Program Analysis

CrayPat is also capable of performing Automatic Program Analysis, and determining which `pat_build` options are mostly likely to produce meaningful data from your program. For more information about using Automatic Program Analysis, see Using Automatic Program Analysis on page 21.

1.1.3 Running the Program and Collecting Data

Instrumented programs are executed in exactly the same way as any other program; either by using the `aprun` command if your site permits interactive sessions or by using your system's batch commands.

When working on a Cray XT system, always pay attention to your file system mount points. While it may be possible to execute a program on a login node or while mounted on the `ufs` file system, this generally does not produce meaningful data. Instead, always run instrumented programs on compute nodes and while mounted on a high-performance file system that supports record locking, such as the Lustre file system.

CrayPat supports more than fifty optional run time environment variables that enable you to control instrumented program behavior and data collection during execution. For example, if you use the C shell and want to collect data in detail rather than in aggregate, consider setting the PAT_RT_SUMMARY environment variable to 0 (off) before launching your program.

```
/lus/nid00008> setenv PAT_RT_SUMMARY 0
```

Doing so can nearly double the amount of data available in Cray Apprentice2, but at the cost of larger data file sizes and increased overhead.

The CrayPat run time environment variables are documented in the intro_craypat(1) man page and discussed in Chapter 3, Using the CrayPat Run Time Environment on page 31.

1.1.4 Analyzing the Results

Assuming your instrumented program runs to completion or planned termination, CrayPat outputs one or more data files. The exact number, location, and content of the data file(s) will vary depending on the nature of your program, the type of experiment for which it was instrumented, and the run time environment variable settings in effect at the time of program execution.

All initial data files are output in .xf format, with a generated file name consisting of your original program name, plus pat, plus the execution process ID number, plus a code string indicating the type of data contained within the file. Depending on the program run and the types of data collected, CrayPat output may consist of either a single .xf data file or a directory containing multiple .xf data files.

To begin analyzing the captured data, use the pat_report command. In simplest form, it looks like this:

```
/lus/nid00008> pat_report myprog+pat+PIDem-n.xf
```

The pat_report command accepts either a file or directory name as input and processes the .xf file(s) to generate a text report. In addition, it also exports the .xf data to a single .ap2 file, which is both a self-contained archive that can be reopened later using the pat_report command and the exported-data file format used by Cray Apprentice2.

The pat_report command provides more than thirty predefined report templates, as well as a large variety of user-configurable options. These reports and options are documented in the pat_report(1) man page and discussed in Chapter 4, Using pat_report on page 41.

Note: If you are upgrading from an earlier version of CrayPat, see Upgrading from Earlier Versions on page 14 for important information about data file compatibility.

1.1.5 Online Help

The CrayPat man pages, online help, and FAQ are available only when the `xt-craypat` module is loaded.

The CrayPat commands, options, and environment variables are documented in the following man pages:

- `intro_craypat`(1) — basic usage and environment variables
- `pat_build`(1) — instrumenting options and API usage
- `hwpc`(3) — optional hardware counter groups that can be used with `pat_build`
- `pat_report`(1) — reporting and data-export options

In addition, CrayPat also includes an extensive online help system, which features many examples and the answers to many frequently asked questions. To access the help system, enter this command:

```
> pat_help
```

The `pat_help` command accepts options. For example, to jump directly into the FAQ, enter this command:

```
> pat_help FAQ
```

Once the help system is launched, navigation is by one-key commands (e.g., `/` to return to the top-level menu) and text menus. It is not necessary to enter entire words to make a selection from a text menu; only the significant letters are required. For example, to select "Building Applications" from the **FAQ** menu, it is sufficient to enter **`Buil`**.

Help system usage is documented further in the `pat_help`(1) man page.

1.1.5.1 Reference Files

When the CrayPat module is loaded, the environment variable `CRAYPAT_ROOT` is defined. Advanced users will find the files in `$CRAYPAT_ROOT/lib` and `$CRAYPAT_ROOT/include` useful. The `/lib` directory contains the predefined trace group definitions (see Using Predefined Trace Groups on page 18) and build directives (see Advanced Users: Environment Variables and Build Directives on page 22), while the `/include` directory contains the files used with the CrayPat API (see Advanced Users: The CrayPat API on page 25).

1.1.5.2 PAPI

CrayPat uses PAPI, the Performance API. This interface is normally transparent to the user. However, if you want more information about PAPI, see the `intro_papi`(3) and `papi_counters`(5) man pages, as well as the *PAPI Programmer's Reference* and *PAPI User's Guide*.

Additional information is available through the PAPI website at http://icl.cs.utk.edu/papi/.

1.1.6 Upgrading from Earlier Versions

If you are upgrading from an earlier version of CrayPat, be advised that file compatibility is not maintained between versions. Programs instrumented using earlier versions of CrayPat must be recompiled, relinked, and reinstrumented using CrayPat 5.0. Likewise, `.xf` and `.ap2` data files created using earlier versions of CrayPat cannot be read using the release 5.0 versions of `pat_report` or Cray Apprentice2, nor can data files created using release 5.0 be read using earlier versions of `pat_report` or Cray Apprentice2.

If you have upgraded to release 5.0 from an earlier version of CrayPat, the earlier version likely remains on your system in the `/opt/modulefiles/xt-craypat` directory. (This may vary depending on your site's software administration and default version policies.) To revert to the earlier version, use the `module swap` command.

For example, assuming that the current default version is 5.0, to revert from CrayPat 5.0 to CrayPat 4.4 so that you can read an old `.ap2` file, enter this command:

```
> module swap xt-craypat xt-craypat/4.4.0
```

To return to the current default version, reverse the command arguments:

```
> module swap xt-craypat/4.4.0 xt-craypat
```

1.2 Analyzing Data with Cray Apprentice2

Cray Apprentice2 is a separately licensed GUI tool for visualizing and manipulating the performance analysis data captured during program execution. Cray Apprentice2 can display a wide variety of reports and graphs, depending on the type of program being analyzed, the way in which the program was instrumented for data capture, and the data that was collected during program execution.

Cray Apprentice2 is not a component of CrayPat, nor is it restricted to analyzing data generated on any particular Cray system. You do not set up or run performance analysis experiments from within Cray Apprentice2. Rather, you use CrayPat first, to instrument your program and capture performance analysis data, and then use Cray Apprentice2 afterwards to visualize and explore the resulting data files.

The number and appearance of the reports that can be generated using Cray Apprentice2 is determined by the kind and quantity of data captured during program execution, which in turn is determined by the way in which the program was instrumented and the environment variables in effect at the time of program execution. For example, changing the PAT_RT_SUMMARY environment variable to 0 before executing the instrumented program nearly doubles the number of reports available when analyzing the resulting data in Cray Apprentice2.

1.2.1 Loading and Launching Cray Apprentice2

To begin using Cray Apprentice2, load the apprentice2 module:

```
> module load apprentice2
```

You do not need to have the CrayPat module loaded in order to use Cray Apprentice2.

To launch the Cray Apprentice2 application, enter this command:

```
> app2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the app2 command returns an "unable to open display" error, contact your system administrator for help in configuring X Window System hosting.

You can specify an .ap2 data file to be opened when you launch Cray Apprentice2:

```
> app2 my_datafile.ap2 &
```

Otherwise, Cray Apprentice2 opens a file selection window and you can then select the file you want to open.

For more information about using the app2 command, see the app2(1) man page.

1.2.2 Online Help

Cray Apprentice2 release 5.0 features an online help system as well as numerous pop-ups and tool-tips that are displayed by hovering the cursor over an area of interest on a chart or graph. To access the online help system, click the **Help** button, or right-click on any report tab and then select **Panel Help** from the pop-up menu.

Feel free to experiment with the Cray Apprentice2 user interface, and to left- or right-click on any area that looks like it might be interesting. Because Cray Apprentice2 does not write any data files, you cannot corrupt, truncate, or otherwise damage your original .ap2 data file using Cray Apprentice2.

1.2.3 Upgrading from Earlier Versions

If you are upgrading from an earlier version of Cray Apprentice2, be advised that file compatibility is not maintained between versions. Data files created using earlier versions of CrayPat cannot be opened in Cray Apprentice2 release 5.0, nor can data files created using CrayPat release 5.0 be opened in earlier versions of Cray Apprentice2.

If you have upgraded to release 5.0 from an earlier version of Cray Apprentice2, the earlier version likely remains on your system in the `/opt/modulefiles/apprentice2` directory. (This may vary depending on your site's software administration and default version policies.) To revert to the earlier version, use the `module swap` command.

For example, assuming that the current default version is 5.0, to revert from Cray Apprentice2 release 5.0 to release 4.4 so that you can read an old `.ap2` file, enter this command:

```
> module swap apprentice2 apprentice2/4.4.0
```

To return to the current default version, reverse the command arguments:

```
> module swap apprentice2/4.4.0 apprentice2
```

Using pat_build [2]

The pat_build command is the instrumenting component of the CrayPat performance analysis tool. After you load the xt-craypat module and recompile your program, use the pat_build command to instrument your program for data capture.

CrayPat supports two categories of performance analysis experiments: *tracing* experiments, which count some event such as the number of times a specific system call is executed, and asynchronous (*sampling*) experiments, which capture values at specified time intervals or when a specified counter overflows.

The pat_build command is documented in more detail in the pat_build(1) man page. For additional information and examples, see pat_help build.

2.1 Basic Profiling

The easiest way to use the pat_build command is by accepting the defaults.

```
> pat_build myprogram
```

This generates a copy of your original executable that is instrumented for the default experiment, samp_pc_time, an experiment that samples program counters at regular intervals and produces a basic profile of the program's behavior during execution.

A variety of other predefined experiments are available. (See Selecting a Predefined Experiment on page 37.) However, in order to use any of these other experiments, you must first instrument your program for tracing.

2.2 Using Predefined Trace Groups

The easiest way to instrument your program for tracing is by using the `-g` option to specify a predefined trace group.

```
> pat_build -g tracegroup myprogram
```

These trace groups instrument the program to trace all function entry point references belonging to the specified group. Only those function entry points actually executed by the program at run time are traced. The valid trace group names are:

`blacs`	Basic Linear Algebra communication subprograms
`blas`	Basic Linear Algebra subprograms
`caf`	Co-Array Fortran (Cray CCE compiler only)
`ffio`	Flexible File I/O (Cray CCE compiler only)
`fftw`	Fast Fourier Transform library
`hdf5`	Manages extremely large and complex data collections
`heap`	Dynamic heap
`io`	Includes `stdio` and `sysio` groups
`lapack`	Linear Algebra Package
`lustre`	Lustre File System
`math`	ANSI math
`mpi`	MPI
`netcdf`	Network common data form (manages array-oriented scientific data)
`omp`	OpenMP API
`portals`	Lightweight message passing API
`pthreads`	POSIX threads
`scalapack`	Scalable LAPACK
`shmem`	SHMEM
`stdio`	All library functions that accept or return the `FILE*` construct
`sysio`	I/O system calls
`system`	System calls
`upc`	Unified Parallel C (Cray CCE compiler only)

The files that define the predefined trace groups are kept in $CRAYPAT_ROOT/lib. To see exactly which functions are being traced in any given group, examine the Trace files. These files can also be used as templates for creating user-defined tracing files. (See Instrumenting a User-defined List of Functions on page 20.)

2.3 User-defined Tracing

Alternatively, you can use the pat_build command options to instrument specific function entry points, to instrument a user-defined list of function entry points, to block the instrumentation of specific functions, or to create new trace intercept routines.

2.3.1 Enabling Tracing and the CrayPat API

To change the default experiment from sampling to tracing, activate any API calls added to your program, and enable tracing for user-defined functions, use the -w option.

```
> pat_build -w myprogram
```

The -w option has other implications which are discussed in the following sections.

2.3.2 Instrumenting a Single Function

To instrument a specific function by name, use the -T option.

```
> pat_build -T tracefunc myprogram
```

This option applies to all the entry points contained within the predefined function groups that are used with the -g option. If the -w option is specified, user-defined entry points are traced as well. (See Using Predefined Trace Groups on page 18.)

If *tracefunc* contains a slash (/) character, the string is interpreted as a basic regular expression. If regular expressions identify any user-defined entry points, the -w option must also be specified to generate trace wrappers.

2.3.3 Preventing Instrumentation of a Function

To prevent instrumentation of a specific function, use the -T ! option.

```
> pat_build -T !tracefunc myprogram
```

If *tracefunc* begins with an exclamation point (!) character, references to *tracefunc* are not traced.

2.3.4 Instrumenting a User-defined List of Functions

To trace a user-defined list of functions, use the `-t` option.

```
> pat_build -t tracefile myprogram
```

The *tracefile* is a plain ASCII text file listing the functions to be traced. For an example of a *tracefile*, see any of the predefined Trace files in `$CRAYPAT_ROOT/lib`.

To specify user-defined functions, also include the `-w` option.

2.3.5 Creating New Trace Intercept Routines for User Files

To create new trace intercept routines for those function entry points that are defined in the respective source file owned by the user, use the `-u` option.

```
> pat_build -u myprogram
```

To prevent a specific function entry point *entry-point* from being traced, use the `-T!` option.

```
> pat_build -u -T'!entry-point' myprogram
```

2.3.6 Creating New Trace Intercept Routines for Everything

To make tracing the default experiment, activate the CrayPat API, and create new trace intercept routines for those function entry points for which no trace intercept routine already exists, use the `-w` option.

```
> pat_build -w -t tracefile[...] -T symbol[...] myprogram
```

If `-t`, `-T`, or the `trace` build directive are not specified, only those function entry points necessary to support the CrayPat run time library are traced. If `-t`, `-T`, or the `trace` build directive are specified, and `-w` is not specified, only those function points that have pre-existing trace intercept routines are traced.

2.4 Using Automatic Program Analysis

The Automatic Program Analysis feature lets CrayPat suggest how your program should be instrumented, in order to capture the most useful data from the most interesting areas. To use this feature, follow these steps.

1. Instrument the original program.

   $ **pat_build -O apa** *my_program*

   This produces the instrumented executable *my_program*+pat.

2. Run the instrumented executable.

   $ **aprun** *my_program*+pat

   This produces the data file *my_program*+pat+*PID-nodes*dt.xf, which contains basic asynchronously derived program profiling data.

3. Use pat_report to process the data file.

   $ **pat_report** *my_program*+pat+*PID-nodes*dt.xf

   This produces three results:

   - a sampling-based text report to stdout
   - an .ap2 file (*my_program*+pat+*PID-nodes*dt.ap2), which contains both the report data and the associated mapping from addresses to functions and source line numbers
   - an .apa file (*my_program*+pat+*PID-nodes*dt.apa), which contains the pat_build arguments recommended for further performance analysis

4. Reinstrument the program, this time using the .apa file.

   $ **pat_build -O** *my_program*+pat+*PID-nodes*dt.apa

   It is not necessary to specify the program name, as this is specified in the .apa file. Invoking this command produces the new executable, *my_program*+apa, this time instrumented for enhanced tracing analysis.

5. Run the new instrumented executable.

   $ **aprun** *my_program*+apa

   This produces the new data file *my_program*+pat+*PID2-nodes*dt.xf, which contains expanded information tracing the most significant functions in the program.

6. Use `pat_report` to process the new data file.

   `$ pat_report` *my_program*`+pat+`*PID2-nodes*`dt.xf`

   This produces two results.

   - a tracing report to `stdout`
   - an `.ap2` file (*my_program*`+pat+`*PID2-nodes*`dt.ap2`) containing both the report data and the associated mapping from addresses to functions and source line numbers

   For more information about Automatic Program Analysis, see `pat_help APA`.

2.5 Advanced Users: Environment Variables and Build Directives

CrayPat supports a number of environment variables and build directives that enable you to fine-tune the behavior of the `pat_build` command. The following environment variables are currently supported.

`PAT_BUILD_LINK_DIR`

If set, specifies the directory in which the object and archive files can be found.

`PAT_BUILD_NOCLEANUP`

If set, specifies if the directory used for intermediate temporary files is removed when `pat_build` terminates.

`PAT_BUILD_OPTIONS`

If set, specifies the `pat_build` options that are to be evaluated before any options on the command line.

`PAT_BUILD_TRACE_ARCHIVE`

If set to nonzero, archive files writable by the user are eligible to have their function entry points traced when the `-u` option is specified. This is the default behavior. To disable this behavior, set this environment variable to zero.

`PAT_BUILD_VERBOSE`

If set, specifies the detail level of the progress messages related to the instrumentation process. This value corresponds to the number of `-v` options specified.

Build directives are invoked either by using the pat_build -d option to read in a build directives file (by default, $CRAYPAT_ROOT/lib/BuildDirectives), or by using the pat_build -D option to specify individual directives. The following build directives are currently supported. The format of each directive is *dirname=dirvalue*.

force-instr=y | n

By default, the pat_build command does not permit a program to be instrumented if it already has been instrumented by another method. If this directive is set to y, the pat_build command ignores the check for prior instrumentation and attempts to force instrumentation of the program. The other methods of instrumenting a program include:

- the PERFCTR, PFM, or PAPI libraries
- the IOBUF or FPMPI libraries
- GNU profiling or GNU coverage analysis
- MPI profiling functions
- previous use of the pat_build command

Caution: Using this directive to force instrumentation of a previously instrumented program may result in an executable that produces incorrect results, exhibits unpredictable behavior, or generates invalid CrayPat performance analysis data.

invalid=*entry-point*[, *entry-point*...]

Specifies one or more function entry points in the original program that inhibit any instrumentation.

link-fatal=*operand*[, *operand*...]

Specifies one or more operands that, if present in the original link, will prevent the instrumented link from occurring.

link-ignore=*operand*[, *operand*...]

Specifies one or more operands that, if present in the original link, will not be passed down to the instrumented link.

link-ignore-libs=*lib*[, *lib*...]

Specifies one or more object or archive files that, if present in the original link, will not be passed down to the instrumented link.

link-instr=*operand*[, *operand*...]

Specifies one or more operands to include in the instrumented link.

`link-objs`=*ofile*[, *ofile*...]

Specifies one or more object files to include in the instrumented link.

`link-options-file=y|n`

By default, the link that produces the instrumented program inserts `ld` operands not included in the link of the original program inline. If set to `y`, this directive puts the `ld` operands into a file and uses the `ld @file` syntax to include the options. This requires GNU `ld` version 2.6.19 or later.

`rtenv`=*name*=*value*[,*name*=*value*,...]

Embeds the run time environment variable *name* in the instrumented program and sets it to value *value*. If a run time environment variable is set using both this directive and in the execution environment, the *value* set in the execution environment takes precedence and this value is ignored.

For more information about run time environment variables, see the `intro_craypat`(1) man page.

`trace`=*entry-point*[, *entry-point*,...]

Specifies one or more function entry points in the original program to trace. If *entry-point* is preceded by the `!` character, function *entry-point* is not allowed to be traced.

`trace-args=y|n`

Collect and record at run time the values of formal parameters for generated trace intercept routines. The default is `n`.

`trace-complex=y|n`

If set to `y`, generate a wrapper for function entry points that return a complex value. The default is `n`.

`trace-debug`=*strng*[,*strng2*,...]

Add verbose print statements to generated trace intercept routines. The string *strng* identifies part or all of the function entry point name. The print statements are activated at run time when the `PAT_RT_VERBOSE` environment variable is set to nonzero. This may be helpful if a traced function entry point is suspected of causing a run time error.

`trace-file`=*strng*[,*strng2*,...]

Activate or deactivate tracing of function entry points in a file. The string *strng* identifies part or all of the file name to activate or deactivate. If *strng* is preceded by an exclamation point (`!`) function entry points in the matched file(s) are not traced.

`trace-max`=*n*

The maximum number of function entry points in the original program that can be traced. The default is 1024. Tracing a large number of entry points results in degraded performance of the instrumented program at run time.

`trace-obj-size`=*min,max*

Specifies the minimum and maximum size in bytes of object and archive files to trace.

`trace-skip`=*strng*[,*strng2*,...]

Silently ignore function entry points when processing them for tracing. The string *strng* identifies part or all of the function entry point name.

`trace-text-size`=*min,max*

Specifies the minimum and maximum size in bytes of text sections in user-defined function entry points to trace. This does not apply to entry points defined in the trace function groups.

`varargs=y|n`

If set to `y`, function entry points that accept variable arguments can be traced. The default is `n`.

2.6 Advanced Users: The CrayPat API

There may be times when you want to focus on a certain region within your code, either to reduce sampling overhead, reduce data file size, or because only a particular region or function is of interest. In these cases, use the CrayPat API to insert calls into your program source, to turn data capture on and off at key points during program execution. By using the CrayPat API, it is possible to collect data for specific functions upon entry into and exit from the functions, or even from one or more regions within the body of the function.

The general procedure for using the CrayPat API looks like this.

Procedure 1. Using CrayPat API Calls

1. Load the CrayPat module.

   ```
   > module load xt-craypat
   ```

2. Include the CrayPat API header file in your source code. Header files for both Fortran and C/C++ are provided in `$CRAYPAT_ROOT/include`.

3. Modify your source code to insert API calls where wanted.

4. Compile your code.

5. Use the `pat_build -w` option to build the instrumented executable. Additional functions can also be specified using the `-t` or `-T` options. The `-u` option (see Creating New Trace Intercept Routines for User Files on page 20) may be used, but it is not recommended as it forces `pat_build` to create an entry point for every user-defined function, which may inject excessive tracing overhead and obscure the results for the regions.

6. Run the instrumented program, and use the `pat_report` command to examine the results.

2.6.1 Header Files

CrayPat API calls are supported in both Fortran and C. The include files are found in `$CRAYPAT_ROOT/include`.

The C header file, `pat_api.h`, must be included in your C source code.

The Fortran header files, `pat_apif.h` and `pat_apif77.h`, may be included in your source or used for reference purposes only. The header file `pat_apif.h` can be used only with compilers that accept Fortran 90 constructs such as new-style declarations and interface blocks. The alternative Fortran header file, `pat_apif77.h`, is for use with compilers that do not accept such constructs.

2.6.2 API Calls

The following API calls are supported. All API usage must begin with a PAT_region_begin call and end with a PAT_region_end call. The examples below show C syntax. The Fortran functions are similar.

int PAT_region_begin (int *id*, const char **label*)
int PAT_region_end (int *id*)

Defines the boundaries of a region. For each region, a summary of activity including time and hardware performance counters (if selected) is produced. The argument *id* assigns a numerical value to the region and must be greater than zero. Each *id* must be unique across the entire program.

The argument *label* assigns a character string to the region, allowing for easier identification of the region in the report.

Two run time environment variables affect region processing: PAT_RT_REGION_CALLSTACK and PAT_RT_REGION_MAX. See the intro_craypat(1) man page for more information about these environment variables.

int PAT_record (int *state*)
int PAT_state (int *state*)
int PAT_sampling_state (int *state*)
int PAT_tracing_state (int *state*)

PAT_record controls the state for all threads on the executing PE. As a rule, use PAT_record unless there is a need for different behaviors for sampling and tracing.

If it is necessary to use the lower-level API functions (PAT_sampling_stateor PAT_tracing_state), these control the state for the respective experiment for the executing thread only. The PAT_state API function is similar to the other lower-level API functions, but determines the active experiment itself.

The lower-level API functions change the state of sampling or tracing to *state*, where *state* can have one of the following values:

PAT_STATE_ON

Activates the *state*.

PAT_STATE_OFF

Deactivates the *state*.

PAT_STATE_QUERY

Returns the current value of *state* without changing it.

All other values have no effect on the *state*. The *state* at the time of the call is returned.

`int PAT_trace_user_l (const char *str, int expr, ...)`

Issues a TRACE_USER record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *str* and the arguments, if specified, in addition to other information, including a timestamp.

Returns the value of *expr*.

This function applies to tracing experiments only.

This function is supported for C and C++ programs only, and is not available in Fortran.

`int PAT_trace_user_v (const char *str, int expr, int nargs, long *args)`

Issues a TRACE_USER record into the experiment data file if the expression *expr* evaluates to true. The record contains the identifying string *str* and the arguments, if specified, in addition to other information, including a timestamp.

nargs indicates the number of 64–bit arguments pointed to by *args*. These arguments are included in the TRACE_USER record.

Returns the value of *expr*.

This function applies to tracing experiments only.

`void PAT_trace_user (const char *str)`

Issues a TRACE_USER record containing the identifying string *str* into the experiment data file.

This function applies to tracing experiments only.

`int PAT_trace_function` (`const void` **addr*, `int` *state*)

Activates or deactivates the tracing of the instrumented function indicated by the function entry address *addr*. The argument *state* is the same as state above. Returns nonzero if the function at the entry address was activated or deactivated, otherwise, zero is returned.

This function applies to tracing experiments only.

`int PAT_flush_buffer` (`void`)

Writes all of the recorded contents in the data buffer to the experiment data file for the calling PE and calling thread. The number of bytes written to the experiment data file is returned. After writing the contents, the data buffer is empty and starts to refill. See `intro_craypat`(1) to control the size of the write buffer.

Note: The data collected by the `PAT_trace_user` API functions is not currently shown on any report. Advanced users may want to collect it and extract information from a text dump of the data files.

For more information about CrayPat API usage, see the `pat_build`(1) man page. Additional information and examples are provided in the help system under `pat_help API`.

2.7 Advanced Users: OpenMP

For programs that use the OpenMP programming model, CrayPat can measure the overhead incurred by entering and leaving parallel regions and work-sharing constructs within parallel regions, show per-thread timings and other data, and calculate the load balance across threads for such constructs.

For programs that use both MPI and OpenMP, profiles by default compute load balance across all threads in all ranks, but you can also see load balances for each programming model separately. For more information about reporting load balance by programming model, see the `pat_report`(1) man page.

The Cray CCE compiler automatically inserts calls to trace points in the CrayPat run time library in order to support the required CrayPat measurements.

PGI compiler release 7.2.0 or later automatically inserts calls to trace points. For all other compilers, including earlier releases of the PGI compiler suite, the user is responsible for inserting API calls.

The following C functions are used to instrument OpenMP constructs for compilers that do not support automatic instrumentation. Fortran subroutines with the same names are also available.

```
void PAT_omp_parallel_enter (void);
void PAT_omp_parallel_exit (void);
void PAT_omp_parallel_begin (void);
void PAT_omp_parallel_end (void);
void PAT_omp_loop_enter (void);
void PAT_omp_loop_exit (void);
void PAT_omp_sections_enter (void);
void PAT_omp_sections_exit (void);
void PAT_omp_section_begin (void);
void PAT_omp_section_end (void);
```

Note that the CrayPat OpenMP API does not support combined parallel work-sharing constructs. To instrument such a construct, it must be split into a parallel construct containing a work-sharing construct.

Use of the CrayPat OpenMP API function must satisfy the following requirements.

- If one member of an `_enter/_exit` or `_begin/_end` pair is called, the other must also be called.
- Calls to `_enter` or `_begin` functions must immediately precede the relevant construct. Calls to `_end` or `_exit` functions must immediately follow the relevant construct.
- For a given parallel region, all or none of the four functions with prefix `PAT_omp_parallel` must be called.
- For a given "sections" construct, all or none of the four functions with prefix `PAT_omp_section` must be called.
- A "single" construct should be treated as if it were a "sections" construct consisting of one section.

Using the CrayPat Run Time Environment [3]

The CrayPat run time environment variables communicate directly with an executing instrumented program and affect how data is collected and saved. Detailed descriptions of all run time environment variables are provided in the `intro_craypat`(1) man page. Additional information can be found in the online help system under `pat_help environment`.

This chapter provides a summary of the run time environment variables, and highlights the more commonly used ones and what they are used for.

3.1 Summary

All CrayPat run time environment variable names begin with `PAT_RT_`. Some require discrete values, while others are toggles. In the case of all toggles, a value of `1` is on (enabled) and `0` is off (disabled).

Table 1. Run Time Environment Variables Summary

Variable Name	Short Description	Default
`PAT_RT_BUILD_ENV`	Toggle: use run time environment variables embedded using the `pat_build rtenv` directive.	`1`
`PAT_RT_CALLSTACK`	Specify the depth to which to trace call stacks.	`100`
`PAT_RT_CALLSTACK_BUFFER_SIZE`	Specify the size in bytes of the run time summary buffer used to collect function call stacks.	`4MB`
`PAT_RT_CHECKPOINT`	Set the maximum number of checkpoint states collected.	`32`
`PAT_RT_COMMENT`	Specify string to insert into experiment data files.	unset
`PAT_RT_CONFIG_FILE`	Specify configuration file(s) containing run time environment variables.	unset

Variable Name	Short Description	Default
PAT_RT_DOFORK	Toggle: enable collection of run time data in a new data file for each forked process.	0
PAT_RT_EXIT_AFTER_INIT	Toggle: terminate execution after initialization of the CrayPat run time library.	0
PAT_RT_EXPERIMENT	Specify the performance analysis experiment to perform.	samp_pc_time if instrumented asynchronously, otherwise trace
PAT_RT_EXPFILE_APPEND	Toggle: append experiment data records to existing experiment data file.	0
PAT_RT_EXPFILE_DIR	Specify the directory in which to write the experiment data file.	current execution directory
PAT_RT_EXPFILE_FIFO	Toggle: create data file as named FIFO pipe instead of a regular file.	0
PAT_RT_EXPFILE_MAX	Specify the maximum number of data files created.	256
PAT_RT_EXPFILE_NAME	Specify the base name of the experiment data file.	base name of instrumented executable
PAT_RT_EXPFILE_PES	Specify the individual PEs from which to collect and record data.	all PEs
PAT_RT_EXPFILE_REPLACE	Toggle: enable overwriting of existing experiment data file(s).	0
PAT_RT_EXPFILE_SUFFIX	Specify the default experiment data filename suffix.	.xf
PAT_RT_HEAP_BUFFER_SIZE	Specify the size in bytes of the buffer used to collect dynamic heap information.	2MB
PAT_RT_HWPC	Specify the hardware performance counter groups to be monitored.	unset
PAT_RT_HWPC_DOMAIN	Specify the domain (1, 2, 4) in which hardware performance counters are active.	Ox1

Variable Name	Short Description	Default
PAT_RT_HWPC_FILE	Specify file(s) containing hardware performance counter event specifications.	unset
PAT_RT_HWPC_FILE_GROUP	Specify file(s) containing hardware performance counter group definitions.	unset
PAT_RT_HWPC_MPX	Toggle: enable multiplexing of hardware performance counter events.	0
PAT_RT_HWPC_OVERFLOW	Specify hardware performance counter overflow frequency and interrupt values.	unset
PAT_RT_INTERVAL	Specify the sampling interval in microseconds.	10000
PAT_RT_INTERVAL_TIMER	Specify the type of interval timer (0–2) used for sampling-by-time experiments.	2
PAT_RT_MEMORY_SET	Specify the 64–bit pattern used to initialize all dynamically allocated memory used by CrayPat.	0
PAT_RT_MPI_SYNC	Toggle: measure MPI load imbalance by measuring the time spent in barrier and sync calls before entering the collective.	1 for tracing experiments, 0 for sampling experiments
PAT_RT_OFFSET	Specify the offset in bytes of the starting virtual address in the text segment to begin sampling.	0
PAT_RT_OMP_SYNC_TRIES	Specify the number of sleep intervals performed by OpenMP slave threads waiting for the main thread to complete CrayPat run time library initialization.	100 sleeps at 100,000 microsecond intervals
PAT_RT_OPEN_MAX	Specify the maximum number of file descriptions that can be open simultaneously.	system maximum
PAT_RT_RECORD_API	Toggle: enable recording of data generated by CrayPat API functions.	1
PAT_RT_RECORD_PE	Deprecated: see PAT_RT_EXPFILE_PES.	

Variable Name	Short Description	Default
PAT_RT_RECORD_THREAD	Specify the individual threads to collect data from.	all threads
PAT_RT_REGION_CALLSTACK	Specify the maximum stack depth for CrayPat API functions PAT_region_begin and PAT_region_end.	128
PAT_RT_REGION_MAX	Specify the largest numerical ID that may be used as an argument to CrayPat API functions PAT_region_begin and PAT_region_end.	100
PAT_RT_SAMPLING_MODE	Specify the mode (0–3) in which trace-enhanced sampling operates.	0
PAT_RT_SAMPLING_SIGNAL	Specify the signal issued when an interval timer expires or a hardware counter overflows.	29 (SIGPROF)
PAT_RT_SETUP_SIGNAL_HANDLERS	Toggle: ignore received signals in order to produce a more accurate traceback.	1
PAT_RT_SIZE	Specify the number of contiguous bytes in the text segment to sample.	all bytes in segment
PAT_RT_SUMMARY	Toggle: enable run time summarization and data aggregation.	1
PAT_RT_THREAD_MAX	Specify the maximum number of POSIX or OpenMP threads that can be created for each process.	8
PAT_RT_TRACE_DEPTH	Specify the maximum depth of the run time callstack.	512
PAT_RT_TRACE_EA_TOLERANCE	Specify the number of lowest-order bits ignored when determining if a function entry address is activated for tracing.	2
PAT_RT_TRACE_FUNCTION_ARGS	Specify the maximum number of function entry point argument values recorded each time the function is called.	256
PAT_RT_TRACE_FUNCTION_DISPLAY	Toggle: write the function entry point names that have been instrumented to stdout.	0

Variable Name	Short Description	Default
PAT_RT_TRACE_FUNCTION_LIMITS	Specify instrumented function entry points to be ignored when tracing.	unset
PAT_RT_TRACE_FUNCTION_MAX	Set maximum number of traces generated for a single process.	unlimited
PAT_RT_TRACE_HEAP	Toggle: collect dynamic heap information.	1
PAT_RT_TRACE_HOOKS	Toggle: record trace data for functions containing compiler-generated hooks.	1
PAT_RT_TRACE_LOOPS	Toggle: collect loop information for use with compiler-guided optimization.	1
PAT_RT_TRACE_OVERHEAD	Specify the number of times calling overhead is sampled during program initialization and termination.	100
PAT_RT_TRACE_THRESHOLD_PCT	Set relative time threshold below which function trace records are not kept.	unset
PAT_RT_TRACE_THRESHOLD_TIME	Set absolute time threshold below which function trace records are not kept.	unset
PAT_RT_VALIDATE_SYSCALLS	Toggle: prevent program from executing function calls that interfere with data collection.	1
PAT_RT_VERBOSE	Toggle: show CrayPat run time activity messages.	0
PAT_RT_WRITE_BUFFER_SIZE	Size of single thread data collection buffer in bytes.	8MB

3.2 Common Uses

3.2.1 Controlling Run Time Summarization

Variable: PAT_RT_SUMMARY

Run time summarization is enabled by default. When it is enabled, data is captured in detail, but automatically aggregated and summarized before being saved. This greatly reduces the size of the resulting experiment data files but at the cost of fine-grain detail. Specifically, when running tracing experiments, the formal parameter values, function return values, and call stack information are not saved.

If you want to study your data in detail, and particularly if you want to use Cray Apprentice2 to generate charts and graphs, disable run time summarization by setting PAT_RT_SUMMARY to 0. Doing so can more than double the number of reports available in Cray Apprentice2.

3.2.2 Controlling Data File Size

Depending on the nature of your experiment and the duration of the program run, the data files generated by CrayPat can be quite large. To reduce the files to manageable sizes, considering adjusting the following run time environment variables.

For sampling experiments, try these:

PAT_RT_CALLSTACK

PAT_RT_EXPFILE_PES

PAT_RT_HWPC

PAT_RT_HWPC_OVERFLOW

PAT_RT_INTERVAL

PAT_RT_SUMMARY

PAT_RT_SIZE

For tracing experiments, try these:

PAT_RT_CALLSTACK

PAT_RT_EXPFILE_PES

PAT_RT_HWPC

PAT_RT_RECORD_THREAD

PAT_RT_SUMMARY

PAT_RT_TRACE_FUNCTION_ARGS

PAT_RT_TRACE_FUCNTION_LIMITS

PAT_RT_TRACE_FUNCTION_MAX

PAT_RT_TRACE_THRESHOLD_PCT

PAT_RT_TRACE_THRESHOLD_TIME

3.2.3 Selecting a Predefined Experiment

Variable: PAT_RT_EXPERIMENT

By default, CrayPat instruments programs for a program-counter *sampling* experiment, samp_pc_time, which samples program counters by time and produces a generalized profile of program behavior during execution. However, if any function entry points are instrumented for tracing by using the pat_build -g, -u, -t, -T, -O, or -w options, then the program is instrumented for a *tracing* experiment, which traces calls to the specified function entry point(s).

After your program is instrumented using pat_build, use the PAT_RT_EXPERIMENT environment variable to further specify the type of experiment to be performed.

Note: Samples generated from sampling by time experiments apply to the process as a whole, and not to individual threads. Samples generated from sampling by overflow experiments apply to individual threads.

The valid experiment types are:

samp_pc_time

The default sampling experiment samples the program counters at regular intervals and records the total program time and the absolute and relative times each program counter was recorded. The default sampling interval is 10,000 microseconds by user and system CPU time intervals, but this can be changed using the PAT_RT_INTERVAL and PAT_RT_INTERVAL_TIMER environment variables. Optionally, this experiment also records the values of the hardware performance counters specified using the PAT_RT_HWPC environment variable.

samp_pc_ovfl

This experiment samples the program counters at the overflow of a specified hardware performance counter. The counter and overflow value are specified using the PAT_RT_HWPC_OVERFLOW environment variable. Optionally, this experiment also records the values of the hardware performance counters specified using the PAT_RT_HWPC environment variable. The default overflow counter is cycles and the default overflow frequency equates to an interval of 1,000 microseconds.

samp_cs_time

This experiment is similar to the samp_pc_time experiment, but samples the call stack at the specified interval and returns the total program time and the absolute and relative times each call stack counter was recorded.

`samp_cs_ovfl`

This experiment is similar to the `samp_pc_ovfl` experiment but samples the call stack.

`samp_ru_time`

This experiment is similar to the `samp_pc_time` experiment but samples system resources.

`samp_ru_ovfl`

This experiment is similar to the `samp_pc_ovfl` experiment but samples system resources.

`samp_heap_time`

This experiment is similar to the `samp_pc_time` experiment but samples dynamic heap memory management statistics.

`samp_heap_ovfl`

This experiment is similar to the `samp_pc_time` experiment but samples dynamic heap memory management statistics.

`trace` Tracing experiments trace the function entry points that were specified using the `pat_build -g`, `-u`, `-t`, `-T`, `-O`, or `-w` options and record entry into and exit from the specified functions. Only true function calls can be traced; function calls that are inlined by the compiler or that have local scope in a compilation unit cannot be traced. The behavior of tracing experiments is also affected by the `PAT_RT_TRACE_DEPTH`, `PAT_RT_TRACE_EA_TOLERANCE`, `PAT_RT_TRACE_FUNCTION_ARGS`, `PAT_RT_TRACE_FUNCTION_DISPLAY`, and `PAT_RT_TRACE_FUNCTION_LIMITS` environment variables, all of which are described in more detail in the `intro_craypat`(1) man page.

Note: If a program is instrumented for tracing and then you use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is performed.

3.2.3.1 Trace-enhanced Sampling

Environment variable: `PAT_RT_SAMPLING_MODE`

If you use `pat_build` to instrument a program for a tracing experiment and then use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is enabled and affects both user-defined functions and predefined function groups.

Trace-enhanced sampling is affected by the PAT_RT_SAMPLING_MODE environment variable. This variable can have one of the following values:

0	Ignore trace-enhanced sampling. Perform a normal tracing experiment. (Default)
1	Enable raw sampling. Any traced entry points present in the instrumented program are ignored.
2	Enable focused sampling. Only traced entry points and the functions they call are sampled.
3	Enable bubble sampling. Traced entry points and any functions they call return a sample program counter address mapped to the trace entry point.

Trace-enhanced sampling is also affected by the PAT_RT_SAMPLING_SIGNAL environment variable. This variable can be used to specify the signal that is issued when an interval timer expires or a hardware counter overflows. The default value is 29 (SIGPROF).

3.2.4 Measuring MPI Load Imbalance

Environment variable: PAT_RT_MPI_SYNC

In MPI programs, time spent waiting at a barrier before entering a collective can be a significant indication of load imbalance. The PAT_RT_MPI_SYNC environment variable, if set, causes the trace wrapper for each collective subroutine to measure the time spent waiting at the barrier call before entering the collective. This time is reported by pat_report in the function group MPI_SYNC, which is separate from the MPI function group, which shows the time actually spent in the collective.

This environment variable affects tracing experiments only and is set on by default.

3.2.5 Monitoring Hardware Counters

Environment variable: PAT_RT_HWPC

Use this environment variable to specify hardware counters to be monitored while performing tracing experiments. The easiest way to use this feature is by specifying the ID number of one of the predefined hardware counter groups; these groups and their meanings vary depending on your system's processor architecture and are defined in the hwpc(3) man page.

More adventurous users may want to load the PAPI module and then use this environment variable to specify one or more hardware counters by PAPI name. To load the PAPI module, enter this command:

```
> module load xt-papi
```

Then use the `papi_avail` and `papi_native_avail` commands to explore the list of counters available on your system. For more information about using PAPI, see the `intro_papi`(3), `papi_avail`(1), and `papi_native_avail`(1) man pages.

The behavior of the `PAT_RT_HWPC` environment variable is also affected by the `PAT_RT_HWPC_DOMAIN`, `PAT_RT_HWPC_FILE`, `PAT_RT_HWPC_FILE_GROUP`, and `PAT_RT_HWPC_OVERFLOW` environment variables. All of these are described in detail in the `intro_craypat`(1) man page.

Using pat_report [4]

The pat_report command is the reporting component of the CrayPat performance analysis tool. After you use the pat_build command to instrument your program, set the run time environment variables as desired, and then execute your program, use the pat_report command to generate text reports from the resulting data and export the data for use in other applications.

The pat_report command is documented in detail in the pat_report(1) man page. Additional information can be found in the online help system under pat_help report.

4.1 Using Data Files

The data files generated by CrayPat vary depending on the type of program being analyzed, the type of experiment for which the program was instrumented, and the run time environment variables in effect at the time the program was executed. In general, the successful execution of an instrumented program produces one or more .xf files, which contain the data captured during program execution.

Unless specified otherwise using run time environment variables, these file names have the format *a.out*+pat+*PID*-*NIDe[m]*.xf, where:

a.out The name of the instrumented executable.

PID The process ID assigned to the instrumented executable at run time.

NID The physical node ID upon which the rank zero process was executed.

e The type of experiment performed, either s for sampling or t for tracing.

m An optional code indicating other special characteristics of the program that produced the data file. These can be:

- d The data was generated by a distributed memory process such as MPI, SHMEM, UPC, or CAF.
- f The data was generated by a forked process.
- o The data was generated by OpenMP.
- t The data was generated by POSIX threads.

Use the pat_report command to process the information in individual .xf files or directories containing .xf files. Upon execution, pat_report automatically generates an .ap2 file, which is both a self-contained archive that can be reopened later using the pat_report command and the exported-data file format used by Cray Apprentice2.

Note: If the executable was instrumented with the pat_build -O apa option, running pat_report on the .xf file(s) also produces an .apa file, which is the file used by Automatic Program Analysis. See Using Automatic Program Analysis on page 21.

4.2 Producing Reports

To generate a report, use the pat_report command to process your .xf file or directory containing .xf files.

```
> pat_report a.out+pat+PIDe[m]-n.xf
```

The complete syntax of the pat_report command is documented in the pat_report(1) man page.

Note: Running pat_report automatically generates an .ap2 file, which is both a self-contained archive that can be reopened later using the pat_report command and the exported-data file format used by Cray Apprentice2. Also, if the executable was instrumented with the pat_build -O apa option, running pat_report on the .xf file(s) produces an .apa file, which is the file used by Automatic Program Analysis. See Using Automatic Program Analysis on page 21.

The pat_report command is a powerful report generator with a wide range of user-configurable options. However, the reports that can be generated are first and foremost dependent on the kind and quantity of data captured during program execution. For example, if a report does not seem to show the level of detail you are seeking when viewed in Cray Apprentice2, consider re-running your program with PAT_RT_SUMMARY set to zero (disabled).

4.2.1 Using Predefined Reports

The easiest way to use pat_report is by using the -O to specify one of the predefined reports. For example, enter this command to see a top-down view of the calltree.

```
> pat_report -O calltree datafile.xf
```

The predefined reports currently available are:

profile Show data by function name only.

callers (or ca)

Show function callers (bottom-up view).

`calltree` (or `ct`)

Show calltree (top-down view).

`ca+src` Show line numbers in callers.

`ct+src` Show line numbers in calltree.

`heap` Implies `heap_program`. `heap_hiwater`, and `heap_leaks`. Instrumented programs must be built using the `pat_build -g heap` option in order to show `heap_hiwater` and `heap_leaks` information.

`heap_program`

Compare heap usage at the start and end of the program, showing heap space used and free at the start, and unfreed space and fragmentation at the end.

`heap_hiwater`

If the `pat_build -g heap` option was used to instrument the program, this report option shows the heap usage "high water" mark, the total number of allocations and frees, and the number and total size of objects allocated but not freed between the start and end of the program.

`heap_leaks` If the `pat_build -g heap` option was used to instrument the program, this report option shows the largest unfreed objects by call site of allocation and PE number.

`load_balance`

Implies `load_balance_program`, `load_balance_group`, and `load_balance_function`. Show PEs with maximum, minimum, and median times.

`load_balance_program`
`load_balance_group`
`load_balance_function`

For the whole program, groups, or functions, respectively, show the `imb_time` (difference between maximum and average time across PEs) in seconds and the `imb_time%` (`imb_time/max_time * NumPEs/(NumPEs - 1)`). For example, an imbalance of 100% for a function means that only one PE spent time in that function.

`load_balance_cm`

If the `pat_build -g mpi` option was used to instrument the program, this report option shows the load balance by group with collective-message statistics.

`load_balance_sm`

If the `pat_build -g mpi` option was used to instrument the program, this report option shows the load balance by group with sent-message statistics.

`loops` If the compiler `-h profile_generate` option was used when compiling and linking the program, display loop count and optimization guidance information.

`mpi_callers`

Show MPI sent- and collective-message statistics.

`mpi_sm_callers`

Show MPI sent-message statistics.

`mpi_coll_callers`

Show MPI collective-message statistics.

`mpi_dest_bytes`

Show MPI bin statistics as total bytes.

`mpi_dest_counts`

Show MPI bin statistics as counts of messages.

`mpi_sm_rank_order`

Uses sent message data from tracing MPI functions to generate suggested MPI rank order information. Requires the program to be instrumented using the `pat_build -g mpi` option.

`mpi_rank_order`

Uses time in user functions, or alternatively, any other metric specified by using the `-s mro_metric` options, to generate suggested MPI rank order information.

`profile_pe.th`

Show the imbalance over the set of all threads in the program.

`profile_pe_th`

Show the imbalance over PEs of maximum thread times.

`profile_th_pe`

For each thread, show the imbalance over PEs.

`program_time`

Shows which PEs took the maximum, median, and minimum time for the whole program.

`read_stats`
`write_stats`

If the `pat_build -g io` option was used to instrument the program, these options show the I/O statistics by filename and by PE, with maximum, median, and minimum I/O times.

`samp_profile+src`

Show sampled data by line number with each function.

`thread_times`

For each thread number, show the average of all PE times and the PEs with the minimum, maximum, and median times.

Note: By default, all reports show either no individual PE values or only the PEs having the maximum, median, and minimum values. The suffix `_all` can be appended to any of the above options to show the data for all PEs. For example, the option `load_balance_all` shows the load balance statistics for all PEs involved in program execution. Use this option with caution, as it can yield very large reports.

4.2.2 User-defined Reports

In addition to the `-O` predefined report options, the `pat_report` command supports a wide variety of user-configurable options that enable you to create and generate customized reports. These options are described in detail in the `pat_report`(1) man page and examples are provided in the `pat_help` online help system.

If you want to create customized reports, pay particular attention to the `-s`, `-d`, and `-b` options.

`-s` These options define the presentation and appearance of the report, ranging from layout and labels, to formatting details, to setting thresholds that determine whether some data is considered significant enough to be worth displaying.

`-d` These options determine which data appears on the report. The range of data items that can be included also depends on how the program was instrumented, and can include counters, traces, time calculations, mflop counts, heap, I/O, and MPI data. As well, these options enable you to determine how the values that are displayed are calculated.

`-b` These options determine how data is aggregated and labeled in the report summary.

For more information, see the `pat_report`(1) man page. Additional information and examples can be found in the `pat_help` online help system.

4.3 Exporting Data

When you use the `pat_report` command to view an `.xf` file or a directory containing `.xf` files, `pat_report` automatically generates an `.ap2` file, which is a self-contained archive file that can be reopened later using either `pat_report` or Cray Apprentice2. No further work is required in order to export data for use in Cray Apprentice2.

Note: Data file compatibility is not maintained between versions. If you are upgrading from an earlier version, `.ap2` files created with earlier versions cannot be used with release 5.0, nor can files created with release 5.0 be viewed with earlier versions. For more information, see Upgrading from Earlier Versions on page 14.

The `pat_report -f` option also enables you to export data to ASCII text or XML-format files. When used in this manner, `pat_report` functions as a data export tool. The entire data file is converted to the target format, and the `pat_report` filtering and formatting options are ignored.

4.4 Automatic Program Analysis

If your executable was instrumented using the `pat_build -O apa` option, running `pat_report` on the `.xf` data file also produces an `.apa` file containing the recommended parameters for reinstrumenting the program for more detailed performance analysis. For more information about Automatic Program Analysis, see Using Automatic Program Analysis on page 21.

Using Cray Apprentice2 [5]

Cray Apprentice2 is an interactive X Window System tool for visualizing and manipulating performance analysis data captured during program execution.

The number and appearance of the reports that can be generated using Cray Apprentice2 is determined solely by the kind and quantity of data captured during program execution. For example, changing the `PAT_RT_SUMMARY` environment variable to `0` (zero) before executing the instrumented program nearly doubles the number of reports available when analyzing the resulting data in Cray Apprentice2.

5.1 Launching the Program

To begin using Cray Apprentice2, load the `apprentice2` module. If this module is not part of your default work environment, enter the following command to load it:

```
> module load apprentice2
```

Note: You do not need to have the CrayPat module loaded in order to use Cray Apprentice2.

To launch the Cray Apprentice2 application, enter this command:

```
> app2 &
```

Alternatively, you can specify the filename to open on launch:

```
> app2 myfile.ap2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the `app2` command returns an "unable to open display" error, see your system administrator for information about configuring X Window System hosting.

The `app2` command supports two options: `--limit` and `--limit_per_pe`. These options enable you to restrict the amount of data being read in from the data file. Both options recognize the `K`, `M`, and `G` abbreviations for kilo, mega, and giga; for example, to open an `.ap2` data file and limit Cray Apprentice2 to reading in the first 3 million data items, enter this command:

```
> app2 --limit 3M data_file.ap2 & &
```

The `--limit` option sets a global limit on data size. The `--limit_per_pe` sets the limit on a per processing element basis. Depending on the nature of the program being examined and the internal structure of the data file being analyzed, the `--limit_per_pe` is generally preferable, as it preserves data parallelism.

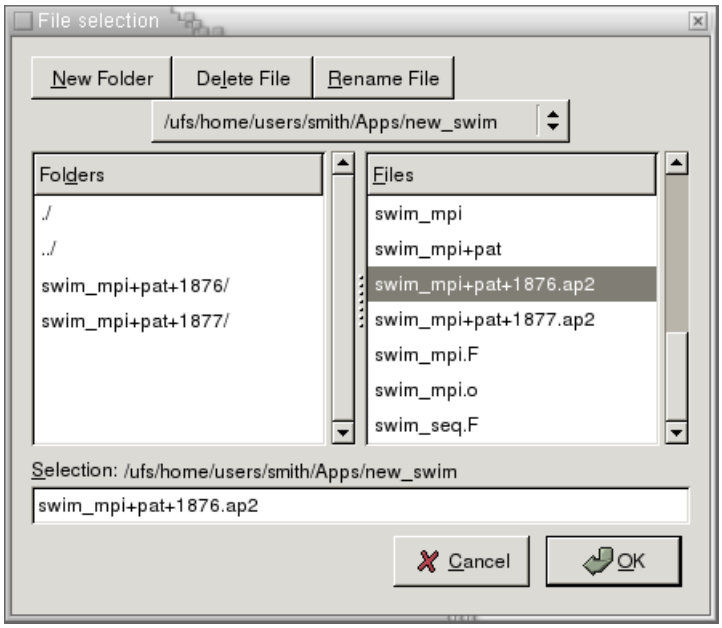
For more information about the `app2` command, see the `app2`(1) man page.

5.2 Opening Data Files

If you specified a valid data file or directory on the `app2` command line, the file or directory is opened and the data is read in, parsed, and displayed.

If you did not specify a data file or directory on the command line, the **File Selection Window** is displayed.

Figure 1. File Selection



Note: As with all other screens in Cray Apprentice2, the exact appearance of the File Selection window varies depending on which version of the Gimp Tool Kit (GTK) is installed on your X Windows System workstation.

After you select a data file, the data is read in. When Cray Apprentice2 finishes parsing the data, the Overview is displayed.

5.3 Basic Navigation

Cray Apprentice2 displays a wide variety of reports, depending on the program being studied, the type of experiment performed, and the data captured during program execution. While the number and content of reports varies, all reports share the following general navigation features.

Figure 2. Screen Navigation

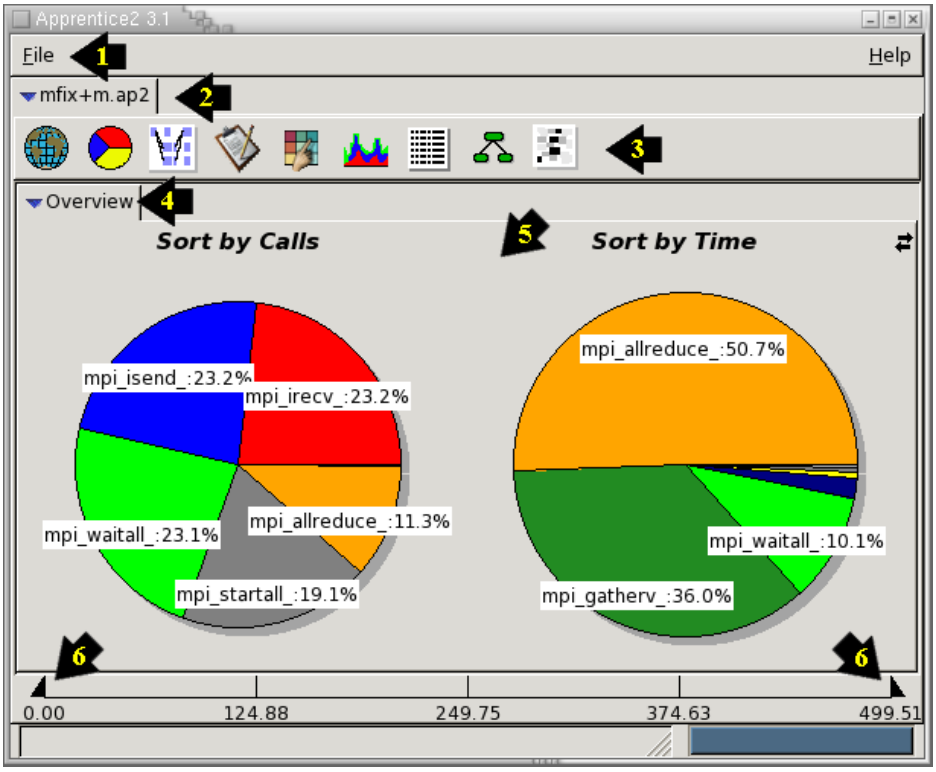


Table 2. Cray Apprentice2 Navigation Functions

Callout	Description
1	The **File menu** enables you to open data files or directories, capture the current screen display to a `.jpg` file, or exit from Cray Apprentice2.
2	The **Data tab** shows the name of the data file currently displayed. You can have multiple data files open simultaneously for side-by-side comparisons of data from different program runs. Click a data tab to bring a data set to the foreground. Right-click the tab for additional options.

Callout	Description
3	The **Report toolbar** show the reports that can be displayed for the data currently selected. Hover the cursor over an individual report icon to display the report name. To view a report, click the icon.
4	The **Report tabs** show the reports that have been displayed thus far for the data currently selected. Click a tab to bring a report to the foreground. Right-click a tab for additional report-specific options.
5	The main display varies depending on the report selected and can be resized to suit your needs. However, most reports feature **pop-up tips** that appear when you allow the cursor to hover over an item, and **active data elements** that display additional information in response to left or right clicks.
6	On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the report window. Move the **caliper points** left or right to restrict or expand the span of time represented by the report. This is a global setting for each data file: moving the caliper points in one report affects all other reports based on the same data, unless those other reports have been detached or frozen.

All report tabs feature **right-click menus**, which display both common options and additional report-specific options. The common right-click menu options are described in Table 3. Report-specific options are described in Viewing Reports on page 51.

Table 3. Common Panel Actions

Option	Description
Screendump	Capture the report or graphic image currently displayed and save it to a `.jpg` file.
Detach Panel	Display the report in a new window.
Remove Panel	Close the window and remove the report tab from the main display.
Freeze Panel	Freeze the report as shown. Subsequent changes to the caliper points do not change the appearance of the frozen report.
Panel Help	Display report-specific help, if available.

5.4 Viewing Reports

The reports Cray Apprentice2 produces vary depending on the types of performance analysis experiments conducted and the data captured during program execution. The report icons indicate which reports are available for the data file currently selected. Not all reports are available for all data.

The following sections describe the individual reports.

5.4.1 Overview Report

The Overview Report is the default report. Whenever you open a data file, this is the first report displayed.

When the Overview Report is displayed, look for:

- In the pie chart on the left, the calls and functions in the program, sorted by the number of times the calls or functions were invoked and expressed as a percentage of the total call volume.
- In the pie chart on the right, the calls and functions in the program, sorted by the amount of time spent performing the calls or functions and expressed as a percentage of the total program execution time.
- Hover the cursor over any section of a pie chart to display a pop-up window containing specific detail about that call or function.
- Right-click the Report Tab to display a pop-up menu that lets you show or hide compute time. Hiding compute time is useful if you want to focus on the communications aspects of the program.
- Alternately, click the Toggle to view this report as a bar graph.

The Overview report is a good general indicator of how much time your program is spending performing which activities and a good place to start looking for load imbalance.

To explore this further, click the function of interest to display a Load Balance Report for the function.

The Load Balance Report shows:

- The load balance information for the function you selected on the Overview Report. This report can be sorted by either PE, Calls, or Time. Click a column heading to sort the report by the values in the selected column.
- The minimum, maximum, and average times spent in this function, as well as standard deviation.
- Hover the cursor over any bar to display PE-specific quantitative detail.

Together, the Overview and Load Balance reports provide a good first look at the behavior of the program during execution and can help you identify opportunities for improving code performance. Look for functions that take a disproportionate amount of total execution time and for PEs that spend considerably more time in a function than other PEs do in the same function. This may indicate a coding error, or it may be the result of a data-based load imbalance.

To further examine load balancing issues, examine the Mosaic and Delta View reports (if available), and look for any communication "hotspots" that involve the PEs identified on the Load Balance Report.

5.4.2 Environment Reports

The Environment Reports provide general information about the conditions under which the data file currently being examined was created. As a rule, this information is useful only when trying to determine whether changes in system configuration have affected program performance.

The Environment Reports consists of four panes. The **Env Vars** pane lists the values of the system environmental variables that were set at the time the program was executed.

> **Note:** This does not include the `pat_build` or CrayPat environment variables that were set at the time of program execution.

The **System Info** pane lists information about the operating system.

The **Resource Limits** pane lists the system resource limits that were in effect at the time the program was executed.

The **Heap Info** pane lists heap usage information.

There are no active data elements or right-click menu options in any of the Environment Reports.

5.4.3 Traffic Report

The Traffic Report shows internal PE-to-PE traffic over time. The information on this report is broken out by communication type (read, write, barrier, and so on). While this report is displayed, you can:

- Hover over an item to display quantitative information.
- Zoom in and out, either by using the zoom buttons or by drawing a box around the area of interest.
- Right-click an area of interest to open a pop-up menu, which enables you to hide the origin or destination of the call or go to the callsite in the source code, if the source file is available.
- Right-click the report tab to access alternate zoom in and out controls, or to filter the communications shown on the report by the duration of the messages.

  Filtering messages by duration is useful if you're only interested in a particular group of messages. For example, to see only the messages that take the most time, move the filter caliper points to define the range you want, and then click the **Apply** button.

The Traffic Report is often quite dense, and typically requires zooming in to reveal meaningful data. Look for large blocks of barriers that are being held up by a single PE. This may indicate that the single PE is waiting for a transfer, or it can also indicate that the rest of the PEs are waiting for that PE to finish a computational piece before continuing.

5.4.4 Mosaic Report

The Mosaic Report depicts the matrix of communications between source and destination PEs, using colored blocks to represent the relative communication times between PEs. By default, this report is based on average communication times. Right-click on the report tab to display a pop-up menu that gives you the choice of basing this report on the Total Calls, Total Time, Average Time, or Maximum Time.

The graph is color-coded. Light green blocks indicates good values, while dark red blocks may indicate problem areas. Hover the cursor over any block to show the actual values associated with that block.

Use the diagonal scrolling buttons in the lower right corner to scroll through the report and look for red "hot spots." These are generally an indication of bad data locality and may represent an opportunity to improve performance by better memory or cache management.

5.4.5 Activity Report

The Activity Report shows communication activity over time, bucketed by logical function such as synchronization. Compute time is not shown.

Look for high levels of usage from one of the function groups, either over the entire duration of the program or during a short span of time that affects other parts of the code. You can use the calipers to filter out the startup and close-out time, or to narrow the data being studied down to a single iteration.

5.4.6 Function Report

The Function Report is a table showing the time spent by function, as both a wall clock time and percentage of total run time. This report also shows the number of calls to the function, the number of call sites in the code that call the function, the extent to which the call is imbalanced, and the potential savings that would result if the function were perfectly balanced.

This is an active report. Click on any column heading to sort the report by that column, in ascending or descending order. In addition, if a source file is listed for a given function, you can click on the function name and open the source file at the point of the call.

Look for routines with high usage, a small number of call sites, and the largest imbalance and potential savings, as these are the often the best places to focus your optimization efforts.

5.4.7 Call Graph

The Call Graph shows the calling structure of the program as it ran and charts the relationship between callers and callees in the program. This report is a good way to get a sense of what is calling what in your program, and how much relative time is being spent where.

Each call site is a separate node on the chart. The relative horizontal size of a node indicates the cumulative time spent in the node's children. The relative vertical size of a node indicates the amount of time being spent performing the computation function in that particular node.

Nodes that contain only callers are green in color.

By default, routines that do not lead to the top routines are hidden.

Nodes that contain callees and represent significant computation time also include stacked bar graphs, which present load-balancing information. The yellow bar in the background shows the maximum time, the purple bar on the left shows the average time, and the cyan (light blue) bar on the right shows the minimum time spent in the function. The larger the yellow area visible within a node, the greater the load imbalance.

While the Call Graph report is displayed, you can:

- Hover the cursor over any node to further display quantitative data for that node.
- Double-click on leaf node to display a Load Balance report for that callsite.
- Right-click the report tab to display a pop-up menu. The options on this menu enable you to change this report so that it shows all times as percentages or actual times, or highlights imbalance percentages and the potential savings from correcting load imbalances. This menu also enables you to filter the report by time, so that only the nodes representing large amounts of time are displayed, or to unhide everything that has been hidden by other options and restore the default display.
- Right-click any node to display another pop-up menu. The options on this menu enable you to hide this node, use this node as the base node (thus hiding all other nodes except this node and its children), jump to this node's caller, or go to the source code, if available.
- Use the zoom control in the lower right corner to change the scale of the graph. This can be useful when you are trying to visualize the overall structure.
- Use the Search control in the lower center to search for a particular node by function name.
- Use the >> toggle in the lower left corner to show or hide an index that lists the functions on the graph by name. When the index is displayed, you can double-click a function name in the index to find that function in the Call Graph.

5.4.8 I/O Reports

The I/O reports are available only if I/O traffic information has been captured. In general, these reports are useful for identifying I/O bottlenecks and conflicts.

There are two I/O reports:

- I/O Overview
- I/O Rates

5.4.8.1 I/O Overview Report

The I/O Overview Report is similar to the Load Balance Report, but shows I/O operations and cumulative times by file descriptor. Like the Load Balance Report, it can help you identify opportunities to improve performance by correcting imbalances in the distribution of I/O work.

This report can be sorted by clicking on the column headings.

5.4.8.2 I/O Rates

The I/O Rates Report is a table listing quantitative information about the program's I/O usage. The report can be sorted by any column, in either ascending or descending order. Click on a column heading to change the way that the report is sorted.

Look for I/O activities that have low average rates and high data volumes. This may be an indicator that the file should be moved to a different file system.

5.4.9 Hardware Reports

The Hardware reports are available only if hardware counter information has been captured. There are two Hardware reports:

- Hardware Counters Overview
- Hardware Counters Plot

5.4.9.1 Hardware Counters Overview Report

The Hardware Counters Overview Report is a bar graph showing hardware counter activity by call and function, for both actual and derived PAPI metrics. While this report is displayed, you can:

- Hover the cursor over a call or function to display quantitative detail.
- Click the "arrowhead" toggles to show or hide more information.

5.4.9.2 Hardware Counters Plot

The Hardware Counters Plot displays hardware counter activity over time and resembles an EKG trace or a seismographic chart. Use this report to look for correlations between different kinds of activity. This report is most useful when you are more interested in knowing *when* a change in activity happens, rather than in the precise quantity of the change.

Look for slopes, trends, and drastic changes across multiple counters. For example, a sudden decrease in floating point operations, accompanied by a sudden increase in L1 cache activity, may indicate a problem with caching or data locality. To zero-in on problem areas, use the calipers to narrow the focus to time-spans of interest on this graph, and then look at other reports to learn what is happening at these times.

To display the value of a specific data point, along with the maximum value, hover the cursor over the area of interest on the chart.

Glossary

blade

1) A field-replaceable physical entity. A Cray XT service blade consists of AMD Opteron sockets, memory, Cray SeaStar chips, PCI-X or PCIe cards, and a blade control processor. A Cray XT compute blade consists of AMD Opteron sockets, memory, Cray SeaStar chips, and a blade control processor. A Cray X2 compute blade consists of eight Cray X2 chips (CPU and network access links), two voltage regulator modules (VRM) per CPU, 32 memory daughter cards, a blade controller for supervision, and a back panel connector. 2) From a system management perspective, a logical grouping of nodes and blade control processor that monitors the nodes on that blade.

Catamount

The operating system kernel developed by Sandia National Laboratories and implemented to run on Cray XT single-core compute nodes. See also *Catamount Virtual Node (CVN)*; *compute node*.

Catamount Virtual Node (CVN)

The Catamount kernel enhanced to run on dual-core Cray XT compute nodes.

CLE

The operating system for Cray XT systems.

compute node

A node that runs application programs. A compute node performs only computation; system services cannot run on compute nodes. Compute nodes run a specified kernel to support either scalar or vector applications. See also *node*; *service node*.

login node

The service node that provides a user interface and services for compiling and running applications.

module

See *blade*.

node

For CLE systems, the logical group of processor(s), memory, and network components acting as a network end point on the system interconnection network.

node ID

A decimal number used to reference each individual node. The node ID (NID) can be mapped to a physical location.

processing element

A processing element is one instance of an executable propagated by the Application Level Placement Scheduler (ALPS).

service node

A node that performs support functions for applications and system services. Service nodes run SUSE LINUX and perform specialized functions. There are six types of predefined service nodes: login, IO, network, boot, database, and syslog.